Cryptographie en Python

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 27 juillet 2009. Dernière mise à jour le 30 juillet 2009

http://www.bortzmeyer.org/python-crypto.html

Quant on fait de la programmation réseau, on a souvent besoin d'utiliser de la cryptographie. En effet, par défaut, les méchants qui sont situés entre les deux machines qui communiquent peuvent écouter tout le trafic et apprendre ainsi des secrets qu'on aurait préféré garder pour soi. Un méchant **actif** peut, dans certains cas, faire encore pire en remplaçant tout ou partie des messages par un contenu de son choix. Comment fait-on de la cryptographie en Python ?

D'abord, un avertissement : il existe plusieurs solutions. Pour des raisons de sécurité, il est recommandé d'utiliser une bibliothèque de haut niveau, qui gère tous les détails elle-même, ce qui donne moins de travail au programmeur et, **surtout**, le dispense de faire une analyse de sécurité de la partie « cryptographie » de son programme, les pièges possibles étant confinés dans la bibliothèque. Parmi les telles bibliothèques, le programmeur Python peut essayer gpgme <http://www.bortzmeyer.org/gpgme.html>, dont le principal avantage est l'interopérabilité avec les autres programmes utilisant la norme OpenPGP (RFC 4880[1]). Ou bien M2crypto <http://chandlerproject.org/Projects/MeTooCrypto>, bâti sur OpenSSL. Ou encore KeyCzar <http://www.keyczar.org/>, que je n'ai pas encore essayé. Une liste détaillée de bibliothèques <http://vermeulen.ca/python-cryptography.html> est disponible (et elle n'indique pas tout ce qui existe).

Mais ce court article est consacré à une autre approche, plus dangereuse, qui nécessite de regarder plus en détail comment ça fonctionne, mais qui permet de réaliser des services de cryptographie exactement comme on le désire. Je vais utiliser le "*Python Cryptography Toolkit*" <https://launchpad.net/pycrypto> (alias PyCrypto), bibliothèque très bien documentée (fichier `pycrypt.ps` dans la distribution) et qui fournit tous les services de base nécessaires pour bâtir une solution de cryptographie.

La bibliothèque figure déjà dans la plupart des systèmes d'exploitation, par exemple, sur Debian, sous le nom de `python-crypto`.

1. Pour voir le RFC de numéro NNN, http://www.ietf.org/rfc/rfcNNN.txt, par exemple http://www.ietf.org/rfc/rfc4880.txt

Je vais développer, avec et sans cryptographie, un client et un serveur pour un protocole trivial : le serveur écoute sur le port 4923, le client se connecte, envoie un texte (encodé en UTF-8) de longueur quelconque et le serveur lui répond par un texte donnant, en anglais, le nombre de caractères du message. Commençons sans la cryptographie : utilisant SocketServer <http://docs.python.org/library/socketserver.html>, voici le serveur (en ligne sur http://www.bortzmeyer.org/files/democrypto-server-plain.py) et le client (en ligne sur http://www.bortzmeyer.org/files/democrypto-client-plain.py). Le mode d'emploi est simple :

```
# Lancement du serveur
% python democrypto-server-plain.py
2009-07-27 21:25:13 INFO Starting server
...
# Lancement du client, on indique l'adresse IP du serveur et le
# message 'tagada'
% python client-plain.py 2a01:e35:8bd9:8bb0:21c:23ff:fe00:6b7f tagada
2009-07-27 21:25:48 INFO Response was You sent 6 characters
...
```

N'importe qui, sur le trajet entre les deux machines (ou même en dehors, grâce à des techniques comme celle de Kapela & Pilosov <http://www.bortzmeyer.org/faille-bgp-2008.html>) pouvait écouter notre message ultra-secret 'tagada'. Il est donc urgent de protéger la communication. Je vais d'abord utiliser la cryptographie symétrique, où les deux parties, le client et le serveur, partagent la même clé secrète. Utilisant le "*Python Cryptography Toolkit*" et l'algorithme de chiffrement AES, on va chiffrer ainsi :

```
coder = AES.new(KEY, AES.MODE_ECB)
...
outf.write("%s" % coder.encrypt(message))
```

et déchiffrer ainsi :

```
self.decoder = AES.new(KEY, AES.MODE_ECB)
...
text = self.server.decoder.decrypt(encrypted_text)
```

Le mode ECB utilisé ici est moins sûr que CBC, également disponible. Il est moins sûr car, avec ECB, un même texte en clair est toujours encodé dans le même texte chiffré, ce qui peut donner des indications à l'adversaire. CBC n'a pas ce défaut mais, dans cet exemple de programme très sommaire, l'état du décodeur n'étant pas réinitialisé à chaque client, CBC, qui nécessite que les deux parties se souviennent de l'état, n'est pas utilisable. Ce problème est une excellente illustration des risques de sécurité qu'on court si on programme à un bas niveau. Avec des bibliothèques comme gpgme <http://www.bortzmeyer.org/gpgme.html>, tout ces risques auraient été gérés en dehors de l'intervention du programmeur.

Autre piège : AES nécessite des données dont la longueur soient des multiples de 16, il va donc falloir faire du remplissage avec des octets nuls (qui ne peuvent pas être présents en UTF-8) :

```
if (len(message) % 16) != 0:
    n = 16 - (len(message) % 16)
    for i in range(0, n):
        message += '\0'
```

ou, plus compact et qui plaira davantager aux Pythonistes (merci à Éric Lebigot pour sa suggestion) :

```
message += '\0' * (-len(message) % 16)
```

Après tous ces détails, voici le serveur (en ligne sur `http://www.bortzmeyer.org/files/democrypto-server-py`) et le client (en ligne sur `http://www.bortzmeyer.org/files/democrypto-client-symmetric-crypto.py`). Le mode d'emploi est exactement le même que précédemment. On peut utiliser Wireshark pour vérifier que le message est bien chiffré.

La cryptographie symétrique marche, elle est simple à programmer, mais elle a des limites. La principale est que les deux parties doivent connaitre la clé (on dit qu'il y a un **secret partagé**). Comme le sait la sagesse populaire, si deux personnes sont au courant, ce n'est plus un secret.

Je vais donc passer en cryptographie asymétrique, dite aussi (à tort) « à clé publique ». En cryptographie asymétrique, chaque clé a deux parties, une publique, qu'on peut distribuer à ses correspondants sans crainte, et une privée, qu'on garde pour soi. Je vais utiliser l'algorithme RSA et faire d'abord un programme qui génère les clés :

```
key = RSA.generate(512, pool.get_bytes)
```

Deux points à noter : la taille de 512 bits pour la clé est bien trop faible, compte-tenu des possibilités de cassage de clés RSA par force brute. Pour un vrai programme, 1024 est probablement un minimum. Et le second point est dans le paramètre `pool.get_bytes`. Le "*Python Cryptography Toolkit*" nécessite que le programmeur fournisse lui-même un générateur aléatoire pour fabriquer la clé. Il n'est pas évident de choisir un bon générateur aléatoire (le RFC 4086 donne de bonnes indications à ce sujet). La plupart des bogues affectant la sécurité des systèmes cryptographique viennent du générateur aléatoire (comme la fameuse bogue Debian `<http://wiki.debian.org/SSLkeys>` due à une modification imprudente d'OpenSSL).

Pour ne pas trop faire souffrir le programmeur, le "*Python Cryptography Toolkit*" fournit un module `Crypto.Util.randpool` qui nous permet d'avoir un générateur tout fait. Encore faut-il penser à l'utiliser (encore un exemple des risques auxquels on s'expose en programmant à bas niveau) :

```
pool = randpool.RandomPool()
# Et la fonction pool.get_bytes fournit ce qu'on veut
```

Attention, `RandomPool` est loin d'être parfait `<http://lists.dlitz.net/pipermail/pycrypto/2008q3/000000.html>` et pourrait disparaitre des prochaines versions du "*Python Cryptography Toolkit*". Trouver un bon générateur aléatoire n'a jamais été facile !

Maintenant que tous ces points sont traités, voici le programme de génération des clés (en ligne sur `http://www.bortzmeyer.org/files/democrypto-create-rsa-key.py`). Il écrit les clés dans des fichiers, le fichier `full.key` contient les parties privées et publiques d'une clé, il doit donc être soigneusement protégé (par exemple, mode 400 sur Unix). Le fichier `public.key` ne contient que la partie publique et peut donc être diffusé largement. On génère donc deux clés :

```
% python democrypto-create-rsa-key.py server
Generating key for server
% python democrypto-create-rsa-key.py client
Generating key for client
```

Le programme en question stocke les clés dans des fichiers sous forme d'objets Python sérialisés avec le module pickle <http://docs.python.org/library/pickle.html>. Ce format est spécifique à Python, si on voulait que d'autres programmes, écrits dans d'autres langages, lisent ces clés, il faudrait définir un format et des programmes d'encodage et de décodage.

Désormais que les clés sont là, on peut programmer le serveur :

```
self.server_key = mykey       # Parties publique et privée
self.client_key = client_key  # Partie publique seulement, le client
       # est le seul à connaitre la partie privée de sa clé
...
data = self.server.server_key.decrypt(encrypted_data)
...
self.wfile.write(self.server.client_key.encrypt(response, None)[0])
```

Comme toujours en cryptographie asymétrique, on chiffre avec une clé et on déchiffre avec une autre. Ici, le serveur connait la clé **publique** du client et chiffre donc la réponse avant de l'envoyer avec cette clé publique. Le client, lui, a chiffré avec la clé publique du serveur, ce qui fait que seul le serveur pourra déchiffre :

```
encrypted_message = server_key.encrypt(message, None)[0]
```

Enfin, voici les codes du serveur (en ligne sur http://www.bortzmeyer.org/files/democrypto-server-a
py) et du client (en ligne sur http://www.bortzmeyer.org/files/democrypto-client-asymmetric-cryp
py). Le mode d'emploi est toujours le même, il faut juste s'assurer que les clés RSA générées plus tôt sont bien disponibles dans le répertoire courant. À noter que, puisque n'importe qui a pu obtenir la clé publique et chiffrer ce message, ce protocole ne fournit pas d'authentification : on ne sait pas qui est en face.

Dans l'exemple précédent, la cryptographie était utilisée pour chiffrer, ce qui préservait la confidentialité des communications mais n'assurait pas l'authentification. RSA permet celle-ci en signant les messages :

```
signature = client_key.sign(message, None)[0]
# On transmet ensuite message et signature, séparés par une chaîne de
# caractères définie dans le protocole
outf.write("%s%s%li" % (message, SEPARATOR, signature))
```

Attention, on signe avec sa propre clé privée. L'autre partie peut alors vérifier cette signature avec la clé publique de son pair :

```
(data, signature_str) = signed_data.split(SEPARATOR)
signature = long(signature_str)
if not self.server.client_key.verify(data, (signature, )):
            raise Exception("Wrong signature in message")
```

En modifiant délibérement le message transmis (par exemple en remplaçant `message` par `message[:-1]`, ce qui en supprime la dernière lettre), on peut vérifier que la signature échoue bien. On a donc à la fois **authentification** (seul le possesseur de la clé privée a pu signer) et **contrôle d'intégrité** (personne n'a modifié le message en cours de route).

Ici, j'ai choisi de signer directement le message. Cela marche mais, la cryptographie asymétrique étant en général beaucoup plus lente que la cryptographie symétrique, la méthode la plus courante pour signer est plutôt de prendre un résumé cryptographique du message (ce qui est une opération relativement rapide) puis de signer ce résumé. Les grands protocoles de cryptographie comme TLS (RFC 5246) fonctionnent tous comme cela.

Voici donc, pour terminer, le serveur (en ligne sur `http://www.bortzmeyer.org/files/democrypto-server-a` `py`) et le client (en ligne sur `http://www.bortzmeyer.org/files/democrypto-client-authentified.py`) avec signature.

Bon courage, amusez-vous bien avec la cryptographie et n'oubliez pas : les programmes présentés ici sont très simplistes, sont livrés sans aucune garantie, comportent des risques connus (comme l'utilisation d'ECB) et certainement d'autres que je ne connais pas.

Un autre article sur ce sujet, très clair et pratique, est ≪ "*Python and cryptography with pycrypto*" <http://www.laurentluce.com/?p=280> ≫. Un exemple d'un grand projet utilisant PyCrypto est ≪ "*Distributed Identity Management in the PGP Web of Trust*" <http://www.seas.upenn.edu/~cse400/CSE400_2005_2006/Margolis/paper.pdf> ≫. Merci à Damien Wyart pour ses bonnes remarques.